Spirit: History and Evolution

Joel de Guzman (joel@boostpro.com)
Hartmut Kaiser (hkaiser@cct.lsu.edu)

Timeline

- 80 - 90s Formative Years
  - Pascal, Recursive Descent, Syntax Diagrams
  - WWW, Text Based Protocols, Small Languages
  - C++
  - Dynamic Spirit
  - Template Metaprogramming, Generative Programming, Expression Templates
- 2000-2001
  - Static Spirit
  - Boost Debut
  - SourceForge
- 2002
  - Phoenix 1.0
  - Spirit X
  - Spirit Meta Lib
  - Spirit Formal Review
  - MPL Formal Review
  - Lambda Formal Review

Timeline

- 2003
  - Wave
  - Spirit 1.8
  - CUJ Article
  - Fusion
- 2004-2005
  - Spirit 2 Design
  - Karma (Tirips)
  - Xpressive Formal Review
  - Wave Formal Review
  - FC++ Formal Review
  - Fusion 2
  - Phoenix 2
- 2006-2007
  - Proto
  - Spirit 2 Development
  - BoostCon
  - Fusion Formal Review
- 2008
  - Proto Formal Review
  - Phoenix Formal Review

Dynamic Spirit

- Library Based
- Recursive descent
- No separate lexer
- Parser objects
  - virtual functions (virtual parse member function)
  - Fine grained one-shot objects
- Composition of Parser objects
  - pointer to base

> *"Experience shows that software development involves lots of parsing … Most of the time, without realizing it, we are coding a parser by hand."*
>
> *Old Spirit Docs ~2000*

SEBNF (Spirit EBNF)

A snippet of the specification of the parser written in itself following the meta-language:

```
grammar         : production*;
production      : ruleID ':' alternative ';';

alternative     : difference ( [|!# difference )*;
difference      : xor ( '-' xor)*;
xor             : intersection ( '^' intersection )*;
intersection    : sequence ( '&' sequence )*;
sequence        : iteration ( iteration )*;

iteration       : action { + | iterator };
iterator        : '@' { integer } { '..' ( integer | '.' ) };
action          : basic { ':' actionID };
```

Match Composition

```
/*=============================================================================
   Spirit Class Library
   Copyright (c) 1999, 2000, Joel de Guzman
   InterXys Inc. All rights reserved.

   Recursive descent parser compiler. Dynamically compiles
   Extended BNF (EBNF) production rules into a working parser.
   The parser acts on the character level and thus obviates
   the need for a separate lexical analyzer stage.

   The generated parser is a hierarchical structure composed
   of Match objects (see Match.hpp). The top-down descent
   traverses the hierarchy, checking each object for a match,
   back-tracking and checking all possible alternatives. Object
   aggregation allows access to any of the step in the recursive
   descent. In addition, named semantic actions can be attached
   to any level within the hierarchy. These actions are C/C++
   functions that will be called if a match is found.

   The complete specification and additional documentation is
   found in the document parser.doc.
=============================================================================*/
```

Match Composition

```
class Match

public:
                        Match();
    virtual             ~Match();
    virtual bool        Parse(Scanner& str) const = 0;

private:
                        Match(Match const&);        //   no copy
    Match&              operator = (Match const&);  //   no assign
};
```

Match Composition

```
/*=============================================================
  AndMatch::Parse(Scanner& str) const

  Return true if both the left and the right match the 'str'.
  The 'str' pointer is advanced only if both the left and right
  result in a full match.
=============================================================*/
bool AndMatch::Parse(Scanner& str) const
{
  Scanner s = str;
  if (Left()->Parse(s) && Right()->Parse(s))
  {
       str = s;
       return true;
  }
  return false;
}
```

Static Spirit

- Hand-coded parser for parsing SEBNF grammars
- Bootstrap! (Write Spirit in Spirit)
  - Possible but not efficient
  - I need a super efficient Parser kernel
  - It must be as efficient as the hand coded parser
- Template Meta… …mplates
  - Jam… …ebruary 1… …1995

…emplates

…ntrodu…

…irei Alexandrescu

"Awesome. I was looking at it right now... the work is fabulous man! Be sure to: 1. Post it to clc++m 2. Write an article on it for CUJ."

Static Spirit CRTP

```
templa
struct
{
   /***

   Deri
   {
      re
   }

   Deri
   {
      re
   }
};
```

The parse member function is conceptual, as opposed to virtual, in the sense that the base class parser does not really have any such member function. Subclasses must provide one. The conceptual base class is a template class parametized by its subclass, which gives it access to its subclass. Whenever a parser is asked to do its task, it delegates the task to its subclass. This process is very similar to how virtual functions work, but the difference is that the member function is bound statically instead of dynamically (run-time bound). James Coplien first popularized this technique of compile-time polymorphism in an article in C++ Report entitled "Curiously Recurring Template Patterns"

The Spirit Parser Library: Inline Parsing in C++
de Guzman, Nuffer
CUJ Sept 2003

Static Spirit

```
template <typename A, typename B>
struct Sequence : public Binary<A, B>, public Parser<Sequence<A, B> > {
    Sequence(A const& a, B const& b) : Binary<A, B>(a, b) {}
    template <typename Scanner>
    Match
    Parse(Scanner& scanner) const
    {
        Scanner s = scanner;
        Match   ma, mb;
        if ((ma = SubjectA().Parse(s)) && (mb = SubjectB().Parse(s)))
        {
            scanner = s;
            return ma + mb;
        }
        return Match();
    }
};
```

Boost Intro

May 21, 2001

Hello there,
Spirit is an object oriented recursive …

I wonder if anyone out there would be interested in such a beast.
If anyones's interested, I'd be very glad to boostify the code
and collaborate with people.

I would appreciate feedback and comments.

Regards,
Joel de Guzman

Boost Intro

Vesa Karvonen

it can make implementing micro parsers an order of magnitude more intuitive and s ared to traditional methods.

Greg Colvin

I've been war nway's ideas with the l library, but giv m to have the time I'm the Spirit work go

Hubert HOLIN

Having the abil t C++ f d be invaluable to m

Douglas Gregor

I think the Spirit C++ parser frame n, readable syntax and that it could form the basis for a great can envision using Spirit C++ syntax and having several back-ends that can generate more efficient parsers, but in-place.

The C Parser

10/16/2001

Hi Joel,

I'm very interested in Spirit since the very first steps of Spirit on the boost list some time ago.

Now I wanted to use Spirit for one of my projects. As an exercise and for getting some experience with your library I implemented a C grammar checker which wasn't so difficult :-) as I thought. The main problems I faced where the need for left recursion free rules and special needs for not matching the shortest possible rules.

Is there any interest in adding this parser as a sample to your library?

BTW I'm using the Intel V5.0.1 compiler with STLPort 4.5 on W2K and there was only one minor change required to compile my parser successfully.

Thanks and regards
Hartmut Kaiser

Semantic Expressions

- Transduction Parsing
- Semantic actions

```
void push(char const* str, char const* end)
{
    my_vec.push_back(std::strtol(str, NULL, 10));
}
```

Semantic Expressions

- Transduction Parsing
- Semantic actions

```
void push(char const* str, char const* end)
{
    my_vec.push_back(std::strtol(str, NULL, 10));
}
```

- ASTs
  - Dan Nuffer
- Attributes
  - int_p → int attribute

```
void push(int attr)
{
    my_vec.push_back(attr);
}
```

Semantic Expressions

```
expression
    =   term[expression.val = arg1]
        >> *(   ('+' >> term[expression.val += arg1])
            |   ('-' >> term[expression.val -= arg1])
            )
    ;

term
    =   factor[term.val = arg1]
        >> *(   ('*' >> factor[term.val *= arg1])
            |   ('/' >> factor[term.val /= arg1])
            )
    ;

factor
    =   ureal_p[factor.val = arg1]
    |   '(' >> expression[factor.val = arg1] >> ')'
    |   ('-' >> factor[factor.val = -arg1])
    |   ('+' >> factor[factor.val = arg1])
    ;
```

Semantic Expressions

- Spirit 1.0 had offline semantic actions
  - E.g. `int_p[assign_a(x)]`
  - There was a whole set of “predefined” actions out of the box.
- Development began with SpiritX
- SpiritX will allow inline “Semantic Expressions”
  - E.g `int_p[var(x) = arg1]`
- Hence became the SE framework (Semantic Expressions Framework)
  - Arguments were always const
  - Supports only 3 arguments (Spirit requires only two)
- SE was generalized and decoupled from Spirit and later became known as Phoenix

Spirit-Meta

- list_p, confix_p, comment_p

- list_p(item, delim)

```
item >> *(delim >> item)
(item - delim) >> *(delim >> (item - delim))
```

```
list_p(item[func], delim)
(item[func] - delim) >> *(delim >> (item[func] - delim))
(item - delim)[func] >> *(delim >> (item - delim)[func])
```

```
list_p(*item, delim)
(*item - delim) >> *(delim >> (*item - delim))
*(item - delim) >> *(delim >> *(item - delim))
```

Spirit-Meta

Spirit mailing list, 2002-10-09,

The grouping parser:

```
expr = add[expr.val = arg1];

add =
    group_d[mul >> '+' >> mul
    [
      add.val = arg1 + arg3
    ]
  | group_d[mul >> '-' >> mul
    [
      adds.val = arg1 - arg3
    ]
  ;
```

```
mul =
    group_d[fact >> '*' >> fact]
      [mul.val = arg1 * arg3]
  | group_d[fact >> '/' >> fact]
      [mul.val = arg1 / arg3]
  | fact[mul.val = arg1]
  ;

fact =
    ureal_p[fact.val = arg1]
  | '('
    >> add[fact.val = arg1
    >> ')'
  | '-' >> add[fact.val = -arg1]
  | '+' >> add[fact.val = arg1]
  ;
```

Spirit-Meta

```
//  The grouped_parser template is the real workhorse behind the group_d parsers,
//  generated if the parser to group is a binary parser.
//
//  The main work this template does, is to attach a special actor as a
//  semantic action recursively to all leaf parsers of the original binary
//  parser. This is done by the help of the post_order parser traversal
//  algorithm. So during the parsing process every leaf parser calls the
//  corresponding operator() functions of its respective actor, providing it
//  with its parser result.
//
//  The attached actors are constructed such, that they assign the parser
//  result values to the corresponding member of the overall result tuple.
//
//  The correct tuple member number is computed during the post_order parser
//  traversal process such, that the leaf parsers are numbered from left to
//  right. Such the most left leaf parser result is accessible as 'arg1' inside
//  the sematic action code attached to the grouped parser, the second leaf
//  parser result is accessible 'arg2' and so on.
//
//  After a successful match this tuple value is fed into the usual
//  Spirit semantic action mechanics and the tuple members are available from
//  inside the semantic action code attached to the grouped parser.
```

Spirit-Meta

- Transformation of parser expressions
  - Traversal of parser expressions

```
template <typename TransformT, typename TupleT>
struct group_transform_policies
  : public group_transform_plain_policy<TransformT, TupleT>,
    public unary_identity_policy<TransformT>,
    public action_identity_policy<TransformT>,
    public binary_identity_policy<TransformT>
{
    group_transform_policies(TupleT &tuple_)
      : group_transform_plain_policy<
            TransformT, TupleT>(tuple_)
    {}
};
```

  - Reconstruct a different parser based on context, if needed
- Each parser needed corresponding parser factory template allowing to reconstruct the original parser component

Boost Formal Review!!!

Ready

Get set

Need I say more? ☺

Boost Formal Review

- A year after Spirit's initial Boost introduction (May 2001 – October 2002), hard work paid off
- Spirit was accepted into Boost
- Review manager: John Maddock
- Participants: Aleksey Gurtovoy, Andre Hentz, Beman Dawes, Carl Daniel, Christopher Currie, Dan Gohman, Dan Nuffer, Daryle Walker, David Abrahams, David B. Held, Dirk Gerrits, Douglas Gregor, Hartmut Kaiser, Iain K.Hanson, Juan Carlos Arevalo-Baeza, Larry Evans, Martin Wille, Mattias Flodin, Noah Stein, Nuno Lucas, Peter Dimov, Peter Simons, Petr Kocmid, Ross Smith, Scott Kirkwood, Steve Cleary, Thorsten Ottosen, Tom Wenisch, Vladimir Prus

Wave

- Wave is a Standards conformant implementation of the mandated C99/C++98 preprocessor functionality
- Generates sequence of C99/C++ tokens exposed by an `iterator` interface

- Started in 2001 as an experiment in how far Spirit could be taken
  - Initially it was not meant to be high performance
  - Merely a experimental platform
  - Almost no Standards conformant preprocessor available
- First full rewrite in 2003
  - Implemented macro namespaces which lead to some discussions in the C++ committee (removed now)
  - Macro expansion trace, which is still a unique feature
  - First official announcement of Wave on Boost list: 3/8/2003

Wave: Some More History

- In Boost since 2005 (V1.33.0)
  - At this time one of the 2 available fully conformant preprocessors (gcc is the other)
- Got a comprehensive regression test suite with over 400 single unit tests
- Continuous improvement and development
  - Based on user feedback
- A lot of improvements since then
  - Performance now on par with commercial preprocessors (but still slower than gcc or clang)
  - Usability improvements helping to use Wave as a library

Explicit Template Instantiation

```
// header some_huge_template.hpp
template <typename T>
struct some_huge_template
{
    some_huge_template() {/* implementation */}
    // ...
};
```

- Using this template is very demanding in terms of compile time
  - Wave needed 30 minutes to compile on Intel C++ V5
- Difficult to decouple as it pulls a lot of other templated stuff

Explicit Template Instantiation

```
// header some_huge_template.hpp
template <typename T>
struct some_huge_template
{
    some_huge_template();     // function declarations only
};

// header some_huge_template_impl.hpp
template <typename T>
some_huge_template<T>::some_huge_template() {/* implementation */}

// source file some_huge_template_string.cpp
#include "some_huge_template.hpp"
#include "some_huge_template_impl.hpp"

template some_huge_template<std::string>;
```

Wave: Known Applications

- Synopsis
  - A Source-code Introspection Tool (http://synopsis.fresco.org/)
- ROSE
  - A Tool for Building Source-to-Source Translators (http://rosecompiler.org/)
- Zoltán Porkoláb
  - Debugger for C++ Templates
- Hannibal
  - partial C++ parser using Spirit
- Wave tool
  - Full blown preprocessor

- If you know more, please drop me a note

Wave: Boost Intro

- First official announcement of Wave on Boost list: Sat 3/8/2003

- "Wave is a great tool that is getting better every day." (Paul Mensonides)
- "And then there is always Wave (if I haven't mentioned it yet, I am really excited about this work!)." (Aleksey Gurtovoy)
- "Congratulations! This is huge!" (Eric Niebler)

- Paul Mensonides: "I say Chaos!"
  - Strict pp lib: 2/27/2003
  - Chaos-PP: 4/10/2003 (first mentioned)

- Vesa Karvonen: "I say Order!"
  - PP Beta reducer: 4/11/2003
    - Order-PP: 4/27/2003

BACK TO SPIRIT

Attribute Grammars

- Spirit synthesized attributes for primitive parsers
  - int_p
  - symbol_p
- Everything else uses transduction
  - a >> b
  - a | b
- This Presents an awkward situation
  - `int x, y;`
  - `int_p[var(x) = arg1 // OK`
  - `(int_p >> int_p)[var(x) = arg1, var(y) = arg2 // No good`
- We need tuples, but tuples ain't enough
- We need a library like MPL that can deal with values
  - Heterogeneous data structures
  - Algorithms on heterogeneous data structures

Attribute Grammars

- Challenge:
  - (a >> b >> c)[var(x) = arg1, var(y) = arg2, var(z) = arg3]
- Walk both the ET tree and the attribute tuple
  - How?
- Solution:
  - flatten the ET tree → ET sequence
  - Walk the ET and attribute sequences in parallel
    - Data structures:
      - ET sequence [A, B, C]
      - Attribute Sequence: [X, Y, Z]
    - Algorithm: `bool any(et, attr, pred)`
      - pred: return true if “any” of et fails (to parse)
- Fusion!

Fusion

- Initially developed as a proof of concept in 2002
  - Based on MPL and seminal work by Doug Gregor
- Bridges the gap between compile time (MPL) and runtime (STL) using similar concepts (sequences, iterators and algorithms)
- Just the perfect infrastructure for Phoenix2 and Spirit2
  - Both are rewrites extensively using Fusion
- Accepted into Boost in May 2006

Fusion

- Fusion is a library for working with heterogenous collections of data, commonly referred to as tuples. A set of containers (vector, list, set and map) is provided, along with views that provide a transformed presentation of their underlying data. Collectively the containers and views are referred to as sequences, and Fusion has a suite of algorithms that operate upon the various sequence types, using an iterator concept that binds everything together.

- The architecture is modeled after MPL which in turn is modeled after STL. It is named "fusion" because the library is a "fusion" of compile time metaprogramming with runtime programming.

ET Invasion

- A plethora of ETs
  - Bind / Lambda / Phoenix
  - Expressive
  - Spirit
    - Qi
    - Karma
  - ETs Unite!
    - Let all ye _1 and _2 be unified
  - **Proto**

Proto

- Proto is a framework for building Domain Specific Embedded Languages in C++. It provides tools for constructing, type-checking, transforming and executing expression templates
- Boost.Proto eases the development of domain-specific embedded languages (DSELs)
  - Never again shall we write yet another operator overload
  - Never again shall we write an “inordinate amount of unreadable and un-maintainable template mumbo-jumbo.”
- Best of all, all our mini-languages seamlessly inter-operate. Case in point: Qi and Karma

Proto

- April 20, 2005: Proto is born as a major refactorization of Boost.Xpressive
- October 28, 2006: Proto is reborn, this time with a uniform expression types that are POD.
- November 1, 2006: The idea for proto::matches<> and the whole grammar facility is hatched during a discussion with Hartmut Kaiser on the spirit-devel list.
- December 11, 2006: The idea for transforms that decorate grammar rules is born in a private email discussion with Joel de Guzman and Hartmut Kaiser.
- April 4, 2007: Preliminary submission of Proto to Boost.
- April 15, 2007: Boost.Xpressive is ported from Proto compilers to Proto transforms. Support for old Proto compilers is dropped.
- January 11, 2008: Boost.Proto v3 brings separation of grammars and transforms and a "round" lambda syntax for defining transforms in-place.
- March 1, 2008: Proto's Boost review begins.
- April 7, 2008: Proto is accepted into Boost.

Tirips, or Spirit spelled backwards

- First ideas: mid 2004
  - Inspired by StringTemplate library (part of ANTLR)

> Tirips is a library for flexible generation of arbitrary character or token sequences. It's very similar to the Spirit parser library in the sense, that it uses a 'grammar' describing the structure of the expected output. It is based on the idea, that a grammar used for parsing of a certain input sequence may be used to regenerate this input sequence as well.

  - First implementation late 2004, since then developed in sync with the parser library
  - Named *Karma* in early 2005, which prompted the name *Qi* for the parser part

Karma: Generating Output

- Today a symmetric part of Spirit
  - “Yang to Qi’s Yin” (Eric Niebler)
- A versatile tools for almost everything from straight output of primitive data items to creating formatted output for complex data structures

- Why use a tool like Karma for generating output?
  - The more complex the output requirements are the more difficult it is to write maintainable code
  - Karma is fast!
  - Karma is header only, same as whole Spirit

Finally: A Lexer

- We have been discussing this on and off
  - Wave developed techniques to integrate a lexer with Spirit
- Lexertl: Ben Hansons excellent lexer engine
- It is a lexical analyser generator inspired by flex.
- The aim is to support all the features of flex, whilst providing a more modern interface and supporting wide characters.
- Much faster than flex
- Proposed for Boost review
- Spirit.Lex is just a wrapper

Acknowledgements

This version of Spirit is a complete rewrite of the Classic Spirit many people have been contributing to (see below). But there are a couple of people who already managed to help significantly during this rewrite. We would like to express our special acknowledgement to:

- **Eric Niebler** for writing Boost.Proto, without which this rewrite wouldn't have been possible, and helping with examples, advices, and suggestions on how to use Boost.Proto in the best possible way.
- **Ben Hanson** for providing us with an early version of his Lexertl library, which is proposed to be included into Boost (as Boost.Lexer). At the time of this writing the Boost review for this library is still pending.
- **Francois Barel** for his silent but steady work on making and keeping Spirit compatible with all versions of gcc, older and newest ones. He not only contributed subrules to Spirit V2.1, but always keeps an eye on the small details which are so important to make a difference.
- **Andreas Haberstroh** for proof reading the documentation and fixing those non-native-speaker-quirks we managed to introduce into the first versions of the documentation.
- **Chris Hoeppler** for taking up the editorial tasks for the initial version of this documentation together with Andreas Haberstroh. Chris did a lot especially at the last minute when we are about to release.
- **Michael Caisse** also for last minute editing work on the 2.1 release documentation.
- **Tobias Schwinger** for proposing expectation points and GCC port of an early version.
- **Dave Abrahams** as always, for countless advice and help on C++, library development, interfaces, usability and ease of use, for reviewing the code and providing valuable feedback and for always keeping us on our toes.
- **OvermindDL** for his creative ideas on the mailing list helping to resolve even more difficult user problems.
- **Carl Barron** for his early adoption and valuable feedback on the Lexer library forcing us to design a proper API covering all of his use cases. He also contributed an early version of the variadic attribute API for Qi.
- **Daniel James** for improving and maintaining Quickbook, the tool we use for this documentation. Also, for bits and pieces here and there such documentation suggestions and editorial patches.
- **Stephan Menzel** for his early adoption of Qi and Karma and his willingness to invest time to spot bugs which were hard to isolate. Also, for his feedback on the documentation.
- **Ray Burkholder** and **Dainis Polis** for last minute feedback on the documentation.
- **Steve Brandt** for his effort trying to put Qi and Karma to some use while writing a source to source language transformation

Thank You!